Juri Surakka

Työvuorosuunnittelua automatisoivan sääntökoneen toteutus

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

29.4.2016

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Juri Surakka Työvuorosuunnittelua automatisoivan sääntökoneen toteutus 14 sivua 29.4.2016
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Simo Silander Konsultti Jaakko Kyrö Sovellusasiantuntija Mikko Nikkanen

Tässä insinöörityössä suunniteltiin ja toteutettiin työvuorosuunnittelua automatisoiva ohjelmistomoduuli. Työ toteutettiin osana Digia Finland Oy:n verkkopohjaisen työvuorosuunnittelusovelluksen Tempus Plannerin tuotekehitystä. Ohjelmistomoduulin tarkoituksena oli nopeuttaa työvuorosuunnittelua sekä parantaa sen laatua identifioimalla suunnittelun toistuvat toimenpiteet ja antamalla sovelluksen itse huolehtia niistä.

Ohjelmistomoduuli on luonteeltaan sääntökone, ja se toteutettiin modulaarisen sovelluskehityksen periaatteiden mukaisesti sekä noudattaen Test Driven Development -menetelmää. Kumpaakaan menetelmää ei sovellettu sokeasti vaan työssä huomioitiin, että molemmissa on omat ongelmansa.

Modulaarisuuden ansiosta sääntökone ei ole riippuvainen sitä käyttävästä asiakassovelluksesta. Tämä mahdollistaa sääntökoneen itsenäisen sovelluskehityksen ja muutokset siinä eivät heijastu asiakassovellukseen eivätkä asiakassovelluksen muutokset heijastu moduuliin.

Moduulille määriteltiin hyvin spesifit ulkoiset ja sisäiset rajapinnat sekä toiminnallisuus. Asiakassovelluksille se tarjoaa rajapintametodit joiden avulla asiakas tietää, minkälaisia sääntöjä on toteutettu sekä mahdollisuuden evaluoida työvuoroja asiakassovelluksessa parametroiduin sääntöjen avulla. Moduuli myös määrittelee rajapinnan datalle, jota se pystyy käsittelemään.

Tämä insinöörityö on julkinen versio varsinaisesta opinnäytetyöstä, ja osia tästä versiosta on määrätty salaiseksi Digia Finland Oy:n toimesta.

Avainsanat	Modulaarinen arkkitehtuuri, Modulaarisuus, Työvuorosuunnittelu, Test Driven Development, Sääntökone, Groovy

Author(s) Title Number of Pages Date	Juri Surakka Implementing Rule Engine for Work Shift Planning Automation 14 pages 29 April 2016
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Simo Silander, Senior Lecturer Jaakko Kyrö, Consultant Mikko Nikkanen, Software Specialist

The goal of this Bachelor's Thesis was to plan and implement a software module which automates work shift planning. It was implemented during the development of Digia Finland's online work force management application. The purpose of the software module was to streamline work shift planning and improve its quality by identifying the repetitive actions of work shift planning and allowing the software do them for you.

The software module is essentially a rule engine and it was implemented according to the principles of modular software architecture and Test Driven Development. However, neither were followed blindly and their weaknesses were addressed during the development process.

Modularity makes the rule engine loosely coupled with its client. This enables independent development of the module and changes in it do not ripple to the client and vice versa. Modularity also makes the rule engine reusable with any client running on the Java platform.

The public and private interfaces and the functionality of the software module were strictly defined and implemented. The module offers two interfaces for its clients. One to identify what kind of rules are implemented and another to evaluate shifts according to rules parametrized in the client. The module also defines an interface for the data it can process.

This version of this Bachelor's Thesis is the public version of the thesis proper and parts of it are classified by Digia Finland.

Keywords	Modular architecture, Modularity, Work Shift Planning, TDD, Test Driven Development, Rule Engine, Groovy

Sisällys

Lyhenteet

API	Ohjelmointirajapinta, jonka avulla eri sovellukset voivat kommunikoida keskenään. Englanniksi Application Programming Interface.
Git	Lähdekoodin hallintaan tarkoitettu hajautettu versionhallintatyökalu.
GitHub	Verkkopohjainen versionhallintatyökalu, joka toimii käyttöliittymänä Git-versionhallintatyökalun päällä.
JAR	Java-ohjelmointikielen tiedostoformaatti, jossa on paketoitu luokkia ja niiden metatietoja ja resursseja yhteen arkistoon, jota voidaan käyttää useissa eri sovelluksissa. Englanniksi Java Archive.
JSON	Ohjelmointikieliriippumaton dataformaatti tiedon välitykseen, jossa data on jaettu avain-arvo-pareihin. Englanniksi JavaScript Object Notation.
Maven	Koontiautomaatiotyökalu, jolla kuvataan, kuinka sovellus kootaan suoritettavaksi tiedostoksi sekä määrittää sovelluksen riippuvuudet.
Mikropalvelu	Itsenäinen ja tilaton prosessi, joka on vastuussa yhdestä sovelluksen toiminnallisuudesta. Käynnissä useimmiten omalla pilvipalvelimella.
MVC	Sovellusarkkitehtuurityyli jolla sovelluksen toiminnallisuus jaetaan kolmeen eri tasoon: malli (Model), näkymä (View) sekä ohjain (Controller). Näistä jokainen hoitaa omaa vastuutaan. Englanniksi Model-View-Controller.
Tietovarasto	Versionhallinnassa oleva tallennuspaikka, jossa lähdekoodi sekä sen metadata sijaitsevat. Englanniksi repository.

1 Johdanto

Perinteinen työvuorosuunnittelu koostuu pahimmillaan sosiaalisesta valtapelistä, post it -lapuista ja seinään teipattavasta työvuorolistasta. Listaan tehdään muutokset lyijykynällä, se vahvistetaan kuulakärkikynällä ja viedään lopuksi palkanlaskijalle. Logistiikan ja tuotannon ohjaamisessa on jo kuitenkin vuosikymmeniä sitten havaittu, kuinka prosesseja optimoimalla ja automatisoimalla saavutetaan huomattavia kustannussäästöjä. WFM (Work Force Management) -järjestelmät mahdollistavat näiden samojen etujen hyödyntämisen työvuorosuunnittelussa. Modernin tyovuorosuunnitteliun avulla ajankäyttö tehostuu, kun työvuoroja suunnitellaan ennalta määritellyn prosessin mukaisesti isolle joukolle. Automaatio tekee perustyön, ja suunnittelijalle jää tehtäväksi sen tarkastus ja tarvittavat muokkaukset. Automatiikka lisää tasapuolisuutta ja läpinäkyvyyttä entisestään mm. raportoinnin kautta. Se myös vähentää työvuorosuunnitteluun kuluvaa aikaa jopa 90 %. (Nurminen ym. 2015.)

Tämän insinöörityön tavoitteena on suunnitella ja toteuttaa ohjelmistomoduuli, jolla automatisoidaan työvuorosuunnittelua. Työ toteutetaan osana Digia Finland Oy:n verkkopohjaisen työvuorosuunnittelusovelluksen Tempus Plannerin tuotekehitystä. Tempus Planner on osa laajempaa Tempus-tuoteperhettä, johon kuuluu lukuisia erilaisia WFM-järjestelmiä. Tämän työn kohteena olevan ohjelmistomoduulin tarkoituksena on nopeuttaa työvuorosuunnittelua sekä parantaa sen laatua identifioimalla suunnittelun toistuvat sekä mekaaniset toimenpiteet ja antamalla sovelluksen itse huolehtia niistä. Moduuli on luonteeltaan sääntökone.

Seuravassa luvussa avataan käyttötapausten kautta sitä, mitä työvuorosuunnittelun automatisoinnilla tarkoitetaan. Tämän jälkeen käydään läpi, millaisia vaatimuksia toteutettavan ohjelmistomoduulin pitää täyttää. Määrittelyn jälkeen käydään läpi moduulin tekninen suunnittelu eli se millaisiin teknisiin ratkaisuihin ja teknologioihin päädyttiin ja miksi. Määrittelyn ja suunnittelun jälkeisessä luvussa tarkastellaan moduulin toteutusta. Luvussa käydään läpi moduulin rajapinnat, yhden säännön toteutus esimerkkitapauksena ja moduulin vaatimat muutokset sitä käyttävään asiakassovellukseen. Lopuksi käydään läpi, kuinka toteutus täyttää määrittelyn asettamat vaatimukset, sekä esille nousseet jatkokehitystarpeet.

Määrittelystä päävastuu oli tuoteomistaja Janne Ojalalla ja asiakaskohtaisia tapauksia käytiin läpi myös projektipäällikkö Pasi Hakosen kanssa. Teknisessä suunnittelussa ohjausta antoivat ohjelmistoasiantuntija Mikko Nikkanen sekä ohjelmistokonsultti Jaakko Kyrö. Vinkkejä tuli myös vanhemmalta ohjelmistoasiantuntijalta Harri Eroselta. Koodikatselmointien kautta moduulin teossa auttoivat edellä mainittujen lisäksi ohjelmistoinsinööri Tomi Harju sekä vanhempi ohjelmistoinsinööri Matti Väänänen.

2 Ongelmasta ratkaisuksi

Työvuorosuunnittelussa on asiakkaasta riippuen erilaisia tarpeita automatisoinnillle. Organisaatioilla on keskenään poikkeavia tapoja suunnittella työvuoroja sekä eri aloilla on voimassa erilaiset lainsäädännöt. Keskeistä automatisoinnissa on hahmottaa, mitkä osat työvuorosuunnittelusta ovat luonteeltaan sellaisia, jotka ovat mekaanisia ja toistettavissa. Toisin sanoen niistä löytyy sekä *ehdot* että *toiminto.*

Esimerkiksi useat työpaikat ovat kiinni viikonloppuisin, joten työvuoroa ei saa luoda lauantaille tai sunnuntaille. Tämä on toimenpide, josta on helposti löydettävissä ehdot (työvuoro osuu lauantaille tai sunnuntaille) ja toiminto (estä työvuoron luonti). Toinen esimerkki on monilla aloilla voimassa oleva viikkotyöaika. Tällöin annettaessa työntekijälle työvuoroja täytyy pitää huolta, etteivät työehtosopimuksissa määritellyt rajat ylity (tai alitu). Olettaen, että tunnemme työehtosopimuksen, myös tästä on löydettävissä ehdot (työvuoro ylittäisi työntekijälle sallitun viikkotyöajan) ja toiminto (estä työvuoron luonti ja lyhennä luotavaa työvuoroa).

Kolmas esimerkki toistuvasta ja mekaanisesta toimenpiteestä työvuorosuunnittelussa on työntekijän sairasloma. Työntekijän sairastuessa jonkun pitää tehdä hänen sairasloman ajan työvuorot eikä hänen työvuoroja voida suoraan muuttaa sairaslomiksi. Työvuoroja ei voida myöskään suoraan siirtää muille työntekijöille tai miehittämättömiksi, sillä tietoa työntekijän sairaslomasta tarvitaan esimerkiksi palkanlaskennan yhteydessä. Näin ollen työvuorosuunnittelijan tehtäväksi tulisi sairasloman vuoron lisäämisen lisäksi etsiä kaikki sairastuneen työntekijän sairaslomalla osuvat työvuorot ja poistaa ne työntekijältä.

Tämänkaltainen tapaus redusoituu *säännöksi*:

> *Kun työntekijälle luodaan työvuoroa, joka on sairaslomaa, poista työntekijältä kaikki sairasloman ajalle osuvat muut työvuorot*.

Kun säännönmukaisuus on identifioitu, on järkevämpää, että sovellus hoitaa sairastuneen työntekijän mahdollisten työvuorojen etsimisen ja vapauttamisen sen sijaan, että työvuorosuunnittelija itse tekisi näin. Sovelluksen näin tehdessä työvuorosuunnittelu nopeutuu ja sen laatu paranee, koska mahdollisuus inhimillisiin virheisiin poistetaan.

Yksinkertaisin tapa toteuttaa yllä mainitut käyttötapaukset olisi kirjoittaa jokaiselle oma ehtolauseensa. Tämä lähestymistapa kuitenkin kävisi hyvin pian mahdottomaksi ylläpitää sekä äärimmäisen vaikeaksi laajentaa. Järkevämpi lähestymistapa onkin erottaa kaikki toiminnallisuus, joka manipuloi työvuoroja tiettyjen ehtojen mukaan, omaksi moduulikseen. Tällainen moduuli on sääntökone, jossa jokainen käyttötapaus on oma sääntönsä, omine ehtoineen ja toimintoineen (Gang ym. 2010: 1; Ziqin ym. 2012: 2). Sääntökone evaluoi työvuoroja sääntöjä vasten ja ehtojen sopiessa yhteen, se muokkaa työvuoroja toimintonsa määrittelemällä tavalla. Seuraavassa luvussa lähdetään käymään läpi, millaisia vaatimuksia tälle moduulille asetetaan.

3 Määrittely

Tämä luku on määrätty salaiseksi Digia Finland Oy’n toimesta.

4 Suunnittelu

4.1 Modulaarinen ohjelmistokehitys teoriassa

Ohjelmistomoduulin teknisen suunnittelun kantava ajatuksena on, että se toteutetaan *modulaarisen ohjelmistokehityksen* periaatteiden mukaan (katso kuva 2). Työvuorosuunnittelusovellus Tempus Planner on luonteeltaan hyvin monimutkainen järjestelmä, joka tarjoaa hyvin monipuolisesti erilaisia toiminnallisuuksia kuten

palkanlaskentaa, työvuorojen optimointia, vuosilomien hallintaa ja niin edelleen. Tämän lisäksi siihen on integroitu useita muita ulkopuolisia järjestelmiä.

Näin laajat järjestelmät ovat luonnostaan monimutkaisempia kehittää ja ylläpitää kuin pienemmät ja yksinkertaisemmat järjestelmät. Modulaarisella arkkitehtuurilla pyritään pilkkomaan laajempi järjestelmä erillisiin osiin - eli moduuleihin - jotta siitä tulee hallittavampi ja ymmärrettävämpi (Knoernschild 2009a). Modulaarisuuden pyrkimyksenä on edistää moduulien *sisäistä* yhtenäisyyttä sekä niiden *välisiä* löyhiä kytkentöjä (Laplante 2007: 86). Näiden ajatusten taustalla on ohjelmistokehityksen kaksi toisiinsa liittyvää perusideaa: *yhden vastuualueen periaate* (single responsibility principle) sekä *rajapintojen erotteluperiaate* (separation of concerns).

Yhden vastuualueen periaatteella tarkoitetaan sitä, että moduuli on vastuussa vain yhdestä toiminnallisuudesta, eli toisin sanoen se sisältää kaiken loogisella tasolla yhteenkuuluvan logiikan. Mikäli moduulilla olisi useampia vastuita, muutos yhdessä vastuussa todennäköisesti pakottaisi tekemään muutoksia moduuliin kaikkiin vastuisiin. Tämän lisäksi yhden vastuun muutokset voivat jopa estää tai haitata moduulin kykyä suoriutua muista vastuistaan. Tällainen vastuiden kytkentä toisiinsa johtaa haavoittuvaan arkkitehtuuriin ja tuottaa odottamattomia tuloksia, kun moduuliin tehdään muutoksia. (Martin & Martin 2006: 138-139.)

Rajapintojen erotteluperiaate tukee ajatusta siitä, että sovelluksen käyttäytyminen kapseloidaan sellaisiin osiin, joissa yksittäiset moduulit sisältävät kaiken loogisella tasolla yhteenkuuluvan logiikan (Laplante 2007: 86). Näiden moduulien väliset suhteet - eli kytkennät - pyritään tekemään mahdollisimman löyhiksi, eli moduulien toteutukset ovat piilossa toisiltaan. Tästä on useita etuja. Kun eri moduulit eivät tunne toisiaan, muutokset yhdessä moduulissa eivät välity toiseen, mikä pienentää virheiden heijastusvaikutusta. (Laplante 2007: 88.) Modulaarisen arkkitehtuurin tarkoitus on nimenomaan varmistaa, että muutokset sovelluksen yhdessä osassa eivät heikennä sen suoriutumista muista toiminnallisuuksista (Meyer & Webb 2005: 18).

Modulaarisuuden tarkoituksena on tehdä sovelluksen arkkitehtuurista ketterä, jotta se pystyy vastamaan muuttuviin vaatimuksiin nopeasti (Knoernschild 2009b; Knoernschild 2009c). Tämä näkyy siinä, että se parantaa sovelluksen uudelleenkäytettävyyttä. Kun moduulit on toteutettu sisäisesti yhtenäisiksi ja keskenään löyhästi voidaan niitä eri tavoin yhdistelemällä vastata uusiin tarpeisiin hyvin pienellä vaivalla. Myös uusien

toiminnallisuuksien lisääminen helpottuu, koska uuden moduulin lisääminen verrattuna tilanteeseen, jossa bisneslogiikoita ei ole toteutettu modulaarisesti, vaatii vähemmän työtä ja on vähemmän virhealtista. (Meyer & Webb 2005: 18-19.)

Uudelleenkäytettävyyden korostamisella on kuitenkin hintansa. Mitä uudelleenkäytettävämpi moduuli on, sitä vaikeammin käytettävämpi siitä itseasiassa tulee. Tämä johtuu siitä, että uudelleenkäytettävyys vaatii, että moduuli on joustava ja joustavuuden mukana kasvaa moduulin monimutkaisuus. (Knoernschild 2009c.) On myös esitetty, että modulaarisuuden kaksi peruspilaria eli moduulien sisäinen yhtenäisyys sekä niiden väliset löyhät kytkennät ovat usein ristiriidassa keskenään ja että tiukempi kytkentä on usein hyväksyttävä hinta moduulin paremmasta sisäisestä yhtenäisyydestä (Heinemeier Hansson 2014a). Meidän ei pidäkään kategorisesti noudattaa modulaarisuuden periaatteita, vaan välillä kannattaa arvioida, tekeekö niiden noudattaminen arkkitehtuurista itseasiassa haavoittuvamman.

4.2 Modulaarinen ohjelmistokehitys käytännössä

Mitä modulaarinen ohjelmistokehitys sitten tarkoittaa käytännössä tämän insinöörityön kontekstissa? Ensinnäkin määrittelyn pohjalta on selvää, että työvuorosuunnittelua automatisoiva ohjelmistomoduuli on luonteeltaan *sääntökone*. Abstraktilla tasolla sääntökoneella on kaksi perusideaa: ensinnäkin erotetaan toisistaan data ja logiikka (Gang ym. 2010, 1). Työvuorosuunnittelun tapauksessa dataa edustaa työvuoro(t) ja logiikkaa sääntöjen toiminnot. Toisekseen sääntökone erottaa sovelluksen erilaiset toiminnallisuudet toisistaan (Gang ym. 2010: 1; Ziqin ym. 2012: 1-2). Näistä sääntökoneen määritelmistä on helppo nähdä, että se implisiittisesti toteuttaa ja tukee modulaarista ohjelmistokehitystä.

Sääntökoneemme - joka sai nimekseen Shift Assistant - on siis teorian tasolla moduuli. Miten se siirretään käytännön tasolle? Ensimmäinen askel Shift Assistantin modularisoimiseksi on perustaa sille oma erillinen projekti sekä tietovarasto. Tämän käytännön toteutus käydään läpi luvussa 5.1. Koontikonfiguraatio. Tämä yksinkertainen lähtökohta pakottaa moduulin toteutuksen itsenäiseksi, koska se ei tunne sitä käyttävää asiakassovellusta Tempus Planneria eikä sillä käytössä sen resursseja. Näin ollen Tempus Planner ja Shift Assistant ovat toisiinsa löyhästi kytkettyjä.

Erillisestä tietovarastosta seuraa myös, että moduuli on agnostinen sitä käyttävän sovelluksen tietomallista. Tämä aiheuttaa sen, että moduuli ei ole riippuvainen mistään yksittäisestä asiakassovelluksesta, vaan sitä voi käyttää mikä tahansa muukin sovellus Plannerin lisäksi. Tietomalliriippumattomuus tekee moduulistamme uudelleenkäytettävän, mutta sillä on myös kääntöpuolensa: moduulin ja sitä käyttävän asiakassovelluksen väliin joudutaan tekemään oma kerroksensa, joka huolehtii, että ne kykenevät toimimaan yhdessä. Tämä toiminnallisuus toteutettiin Plannerin sisään ja sen toteutus käydään läpi yleisellä tasolla luvussa 5.5. Muutokset asiakassovellukseen.

Shift Assistantin vastuualueena on sääntöjen logiikan määrittäminen sekä niiden soveltaminen työvuoroihin. Shift Assistanti rajapintojen toteutukset käydään läpi luvuissa 5.2. Shift Assistantin rajapintametodit sekä 5.5. Sääntökoneen moottori. Moduuli itse määrittää säännölle sekä työvuorolle omat rajapintansa. Nämä käydään läpi tarkemmin luvuissa 5.3. Säännön (Rule) rajapinta ja 5.4. Työvuoron (Shift) rajapinta.

4.3 Test Driven Development

Sääntökoneen toteutuksessa noudatetaan Test Driven Development (TDD) -menetelmää. Pohjimmiltaan TDD jakaa ohjelmistokehityksen kolmeen vaiheeseen: 1) Kirjoita testi uudelle toiminnallisuudelle 2) Kirjoita toiminnallisuuden koodi niin, että testi menee läpi 3) Refaktoroi testien ja toiminnallisuuden koodi hyvin jäsennellyksi. Näitä vaiheita toistetaan, kunnes haluttu ominaisuus tai sovellus on valmis (Esim. Fowler 2005).

TDD:n noudattamisella on useita etuja: ensinnäkin se takaa, että koodilla on testit (Fowler 2005). Testit itsessään varmistavat, että koodin toiminnallisuuteen ei tule odottamattomia muutoksia, koska testit ajetaan vähintään jokaisen koonnin yhteydessä ja testi(e)n epäonnistuessa tiedetään heti, että tehdyillä muutoksilla on ollut joitain ei-toivottuja vaikutuksia. Tämän lisäksi testit toimivat koodin dokumentaationa. (Shore 2010.) Testit myös mahdollistavat, muutoksen ja tukevat modulaarisen ohjelmistokehityksen pyrkimystä ketteryyteen ja kykyyn vastata muuttuviin vaatimuksiin nopeasti, sillä ilman testejä muutosten tekemisen vaikutuksia on mahdotonta kontrolloida (Martin 2009, 124).

Joidenkin tutkimusten mukaan TDD:n noudattaminen vähentää koodin virheitä sekä parantaa sen laatua (Čaušević ym. 2012: 268; Wilkerson ym. 2012: 549). Jotkut tutkimukset ovat myös esittäneet, että TDD:n noudattaminen parantaisi sovelluskehittäjän tuottavuutta (Bajaj ym. 2015: 1; Wilkerson ym. 2012: 559). Toisissa tutkimuksissa tällaista ei ole kuitenkaan havaittu (Rafique & Misic 2013: 854).

TDD myös pakottaa sovelluskehittäjän ajattelemaan toiminnallisuuden rajapintaa ennen varsinaista toteutusta, mikä puolestaan edesauttaa pitämään ne toisistaan erillään (Fowler 2005). Tämä ohjaa sovelluskehittäjän toteutuksiin, joissa on helppokäyttöiset rajapinnat sen sijaan, että toiminnallisuus olisi toteutettu mahdollisimman helposti (Shore 2010). Tämä tukee hyvin modulaarista ohjelmistokehitystä ja sen ideaa rajapintojen tärkeydestä.

TDD:n on kuitenkin myös kritisoitu aiheuttavan vahinkoa sovelluksen arkkitehtuurille. Pahimmillaan pyrkimys kirjoittaa yksikkötestattavaa koodia luo tarpeettomia abstraktioita sekä ylimääräisiä kerroksia sovellukseen. Pääasiassa kritiikki kuitenkin kohdistuu verkkosovellusten ohjaimien (Controller) sekä näkymien (View) yksikkötestaamiseen. (Heinemeier Hansson 2014a; Heinemeier Hansson 2014b) Vaikka Shift Assistantissa ei ole minkäänlaista ohjainlogiikkaa tai visuaalista komponenttia, pitää kuitenkin huomioida, että samaan tapaan kuin modulaarisuuden periaatteiden kategorisen noudattamisen kanssa, TDD:n noudattaminen liian jäykästi saattaa itse asiassa heikentää sovelluksen arkkitehtuuria.

4.4 Groovy-ohjelmointikieli

Shift Assistant toteutetaan Groovy-ohjelmointikielellä. Groovy on täysin integroitu Javaan ja toimii kaikkialla missä Javakin (Esim. Rocher & Brown 2009: 4; The Groovy Programming language). Näin ollen moduulia voi käyttää mikä tahansa asiakassovellus, joka toimii Java-alustalla.

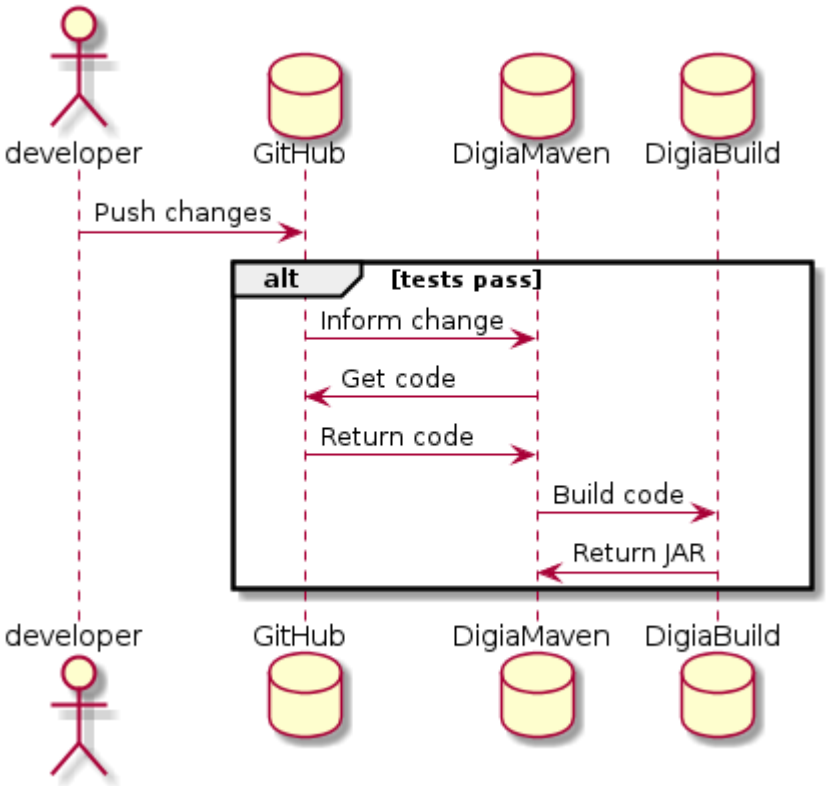
Groovyn valintaa tukee myös se, että se on Javaan verrattuna ilmaisuvoimaisempi, mikä tarkoittaa sitä, että se on syntaksiltaan lyhempi kieli. Tämä johtuu siitä, että Groovy tekee “konepellin alla” paljon sellaista, joka sekä poistaa tarpeetonta toistavaa koodia että tarjoaa sellaisia toiminnallisuuksia, joita Javassa ei ole. (Esim. Rocher & Brown 2009: 545-546; Layka ym. 2013: 7-14.) Tämän lisäksi Groovy soveltuu erityisen hyvin juuri

bisneslogiikan toteuttamiseen sen lyhyen ja helposti luettavan syntaksin ansiosta (Laforge 2016).

5 Toteutus

5.1 Koontikonfiguraatio

Edellisessä luvussa käytiin läpi, kuinka erilliset tietovarastot edistävät modulaarisuutta. Miten tietovarastot (repository) sitten konfiguroitiin? Tempus Plannerin tietovarasto on Digian omassa Git-tietovarastossa, mutta Shift Assistantin tietovarasto pystytettiin GitHubiin. Tämä täysin eri Git-tietovarastojen käyttö johtui siitä, että koko projektin koodikantaa siirrettiin vähitellen omista tietovarastoista GitHubiin. Modulaarisuuden saavuttamiseksi ei vaadita kuitenkaan täysin erillisiä Git-tietovarastoja, vaan keskeinen idea on, että moduulilla on sitä käyttävästä asiakassovelluksesta oma erillinen tietovarasto.



Kuva 1. Shift Assistantin koontikonfiguraatio

Tämän jälkeen Shift Assistantin tietovarasto konfiguroidaan niin, että aina, kun sinne yritettiin puskea uutta lähdekoodia, ajetaan projektin kaikki testit (katso kuva 3). Mikäli yksikin testi epäonnistuu, ei muutoksia pusketa GitHubille asti, ja prosessi päättyy siihen. Mikäli testit läpäistään, konfiguroidaan GitHub-tietovarasto ilmoittamaan Digian omalle Maven-tietovarastolle muutoksista. Tämän jälkeen Digian Maven-tietovarasto hakee uudet lähdekoodit GitHubista ja kokoaa ne Digian koontipalvelimella JAR-moduuliksi. Tämä JAR-moduuli tallennetaan Digian Maven-tietovarastoon.

Lopuksi Tempus Planneriin määritetään Mavenin avulla riippuvuus Shift Assistantiin. Näin ollen niiden välillä on löyhä kytkentä eli Tempus Planner tuntee Shift Assistantin mutta Shift Assistant ei tunne sitä käyttävää asiakassovellusta.

5.2 Shift Assistantin rajapintametodit

Tämä luku on määrätty salaiseksi Digia Finland Oy’n toimesta.

5.3 Säännön (Rule) rajapinta

Tämä luku on määrätty salaiseksi Digia Finland Oy’n toimesta.

5.4 Työvuoron (Shift) rajapinta

Tämä luku on määrätty salaiseksi Digia Finland Oy’n toimesta.

5.5 Sääntökoneen moottori

Tämä luku on määrätty salaiseksi Digia Finland Oy’n toimesta.

5.6 Sairasloma-säännön toteutus

Tämä luku on määrätty salaiseksi Digia Finland Oy’n toimesta.

5.6.1 Testit evaluateRule()-metodin toteutukselle

Tämä luku on määrätty salaiseksi Digia Finland Oy’n toimesta.

5.6.2 EvaluateRule()-metodin toteutus

Tämä luku on määrätty salaiseksi Digia Finland Oy’n toimesta.

5.6.3 Testit applyRule()-metodin toteutukselle

Tämä luku on määrätty salaiseksi Digia Finland Oy’n toimesta.

5.6.4 ApplyRule()-metodin toteutus

Tämä luku on määrätty salaiseksi Digia Finland Oy’n toimesta.

5.7 Muutokset asiakas-sovellukseen

Tämä luku on määrätty salaiseksi Digia Finland Oy’n toimesta.

6 Yhteenveto

Tämän insinöörityön tavoitteena oli suunnitella ja toteuttaa ohjelmistomoduuli jolla automatisoidaan työvuorosuunnittelua. Shift Assistantin ensimmäinen versio toteutettiin maaliskuussa 2015 ja se on ollut tuotantokäytössä huhtikuusta 2015 lähtien.

Moduuli on luonteeltaan sääntökone joka toteutettiin modulaarisen sovelluskehityksen periaatteiden mukaisesti ja noudattaen Test Driven Development-menetelmää. Kumpaakaan menetelmää ei sovellettu sokeasti vaan työssä huomioitiin, että molemmissa on omat ongelmansa. Shift Assistantin toteuttaminen modulaarisesti teki tästä ensimmäisestä toteutuksesta työläämmän, kuin jos sen tarjoama toiminnallisuus olisi toteutettu suoraan asiakassovellukseen. Modulaarisuuden tuomat edut kuitenkin maksavat pian takaisin tämän sijoituksen.

Lähteet

Bajaj Kamini, Patel Hardik, Patel Jeetendra (2015): E*volutionary Software Development using Test Driven Approach*. 2015 International Conference and Workshop on Computing and Communication (IEMCON), 2015, 1-6. IEEE Conference Publications.

Čaušević, Adnan, Punnekkat Sasikumar, Sundmark Daniel (2012): *Quality of Testing in Test Driven Development.* 2012 Eighth International Conference on the Quality of Information and Communications Technology (QUATIC), 2012, 266-271, IEEE Conference Publications.

Fowler Martin (2005): *TestDrivenDevelopment.* <http://martinfowler.com/bliki/TestDrivenDevelopment.html>. Verkkodokumentti. Luettu 22.2.2016.

Gang Zhang, Wenwei Shan, Feng Wang (2010): *Research on the Promotion of Rule Engine Performance.* Intelligent Systems and Application (ISA), 2010 2nd International Workshop, 22-23 May 2010, 1-3. IEEE Conference Publications.

Heinemeier Hansson David (2014a): *Is TDD dead?* Verkko-dokumentti. <http://martinfowler.com/articles/is-tdd-dead/> Luettu 25.2.2016.

Heinemeier Hansson David (2014b): *Test-induced design damage.* Verkkodokumentti <http://david.heinemeierhansson.com/2014/test-induced-design-damage.html>. Luettu 25.2.2016.

Knoernschild Kirk (2009a): *Java Application Architecture. Chapter 1 - Introduction.* <http://www.kirkk.com/modularity/2009/12/chapter-1-introduction/>. Verkkodokumentti Luettu 14.2.2016.

Knoernschild Kirk (2009b): *Java Application Architecture. Chapter 4 - Architecture and modularity.* <http://www.kirkk.com/modularity/2009/12/chapter-4-architecture-and-modularity/>. Verkkodokumentti. Luettu 28.2.2016.

Knoernschild Kirk (2009c): *Java Application Architecture. Chapter 5 - Taming the beast.* <http://www.kirkk.com/modularity/2009/12/chapter-5-taming-the-beast/>. Verkkodokumentti. Luettu 28.2.2016.

Laforge Guillaume (2016): *What are the pros and cons of Groovy?* <https://www.quora.com/What-are-the-pros-and-cons-of-Groovy/answer/Guillaume-Laforge> Verkkodokumentti. Luettu 16.2.2016.

Laplante, Philip (2007): *What every engineer should know about software engineering.* Taylor & Francis. Boca Raton. 2007.

Layka Vishal, Judd Christopher M., Nusairat Joseph Faisal, Shingler Jim (2013): *Beginning Groovy, Grails and Griffon.* Springer Science+Business Media. New York. 2013.

Martin C. Robert (2009): *Clean Code: a handbook of agile software craftsmanship.* Pearson Education Ltd. United States. 2009.

Martin C. Robert, Martin Micah (2006): *Agile Principles, Patterns, and Practices in C#.* Prentice Hall. 2006.

Meyer, M.H. and Webb, P.H. (2005): *Modular, layered architecture: the necessary foundation for effective mass customisation in software*, Int. J. Mass Customisation, Vol. 1, No. 1, 14–36, 2005.

Nurminen Josette, Ojala Janne, Sillanpää Hanna, Tusa Tuukka (2015): *Modernilla työvuorosuunnittelulla hyvinvointia ja tehokkuutta: Lisää tuottavuutta, tyytyväisempää henkilöstöä ja parempia asiakaskokemuksia modernin työvuorosuunnittelun keinoin.* Verkkodokumentti. <http://resources.digia.com/tyovuorosuunnittelu-opas> Luettu 28.3.2016.

Rafique Yahya, Misic Vojislav B. (2013): *The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis.* IEEE Transaction on Software Engineering, 2013, Volume 39, Issue 6, 835-856. IEEE Journals & Maganizes.

Rocher Graeme, Brown Jeff (2009): *The definitive guide to Grails. Second edition.* Springer-Verlag. New York. 2009.

Shore James (2010): *The Art of Agile: Test-Driven development.* Verkkodokumentti. <http://www.jamesshore.com/Agile-Book/test_driven_development.html> Luettu 22.2.2016.

The Grails Framework. https://grails.org/. Verkkodokumentti. Luettu 16.2.2016.

The Groovy Programming language. http://www.groovy-lang.org/. Verkkodokumentti. Luettu 16.2.2016.

Wilkerson Jerod W., Nunamaker Jr. Jay F., Mercer Rick (2012), *Comparing the Defect Reduction Benefits of Code Inspection and Test-Driven Development.* IEEE Transactions on Software Engineering, Volume 38, Issue 3, 2012, 547 - 560. IEEE Journals & Maganizes.

Ziqin Yin, Guojin Zhang, Tengfei Wang, Shijin Li (2012): *Rule engine-based web services composition.* World Automation Congress (WAC), 2012, 1-4, IEEE Conference Publications.